Nonintrusive Snapshots Using Thin Slices

Ajay D. Kshemkalyani and Bin Wu

Computer Science Department, Univ. of Illinois at Chicago, Chicago, IL 60607, USA
{ajayk, bwu}@cs.uic.edu

Abstract. This paper gives an efficient algorithm for recording consistent snapshots of an asynchronous distributed system execution. The nonintrusive algorithm requires \(6(n - 1)\) control messages, where \(n\) is the number of processes. The algorithm has the following properties. (P1) The application messages do not require any changes, not even the use of timestamps. (P2) The application program requires no changes, and in particular, no inhibition is required. (P3) Any process can initiate the snapshot. (P4) The algorithm does not use the message history. A simple and elegant three-phase strategy of uncoordinated observation of local states is used to give a consistent distributed snapshot. Two versions of the algorithm are presented. The first version records consistent process states without requiring FIFO channels. The second version records process states and channel states consistently but requires FIFO channels. The algorithm also gives an efficient way to detect any stable property, which was an unsolved problem under assumptions (P1)-(P4).

1 Problem Definition

A distributed system is modeled as a directed graph \((N, L)\), where \(N\) is the set of processes and \(L\) is the set of links connecting the processes. Let \(n = |N|\) and \(l = |L|\). A distributed snapshot represents a consistent global state of the system. Recording distributed snapshots of an execution is a fundamental problem in asynchronous message-passing systems. Since the seminal algorithm of Chandy and Lamport [3] which is a non-inhibitory algorithm that requires FIFO channels and \(2l\) messages to record a snapshot, plus additional messages to assemble the snapshot, several algorithms have been proposed. A survey is given in [8].

This paper gives an efficient nonintrusive non-inhibitory algorithm for recording consistent snapshots of an asynchronous distributed system execution. The algorithm requires \(6(n - 1)\) control messages and has the following properties.

P1. The application messages require no changes, not even timestamps.
P2. The application program requires no changes, implying no inhibition.
P3. Any process can initiate the snapshot.
P4. The algorithm does not require the log of the message history.

These properties are important for ubiquitous and pervasive computing. A simple and elegant three-phase strategy of uncoordinated observation of local states gives a consistent distributed snapshot. Two versions of the algorithm are
Table 1. Comparison of global snapshot algorithms. The acronym p.b. denotes that control information is piggybacked on the application messages.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Channels required</th>
<th>Non-inhibitory</th>
<th>Application messages unmodified</th>
<th>Number of control messages</th>
<th>Snapshot collection not needed</th>
<th>Message history not used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chandy-Lamport [3]</td>
<td>FIFO</td>
<td>Y</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Specialle-Kearns [16]</td>
<td>FIFO</td>
<td>Y</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Venkatesan [17]</td>
<td>FIFO</td>
<td>Y</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Helary [5] (wave sync.)</td>
<td>FIFO</td>
<td>N</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Helary [5] (wave sync.) non-FIFO</td>
<td>non-FIFO</td>
<td>N</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Lai-Yang [10]</td>
<td>non-FIFO</td>
<td>N (p.b.)</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LRV [12]</td>
<td>non-FIFO</td>
<td>N (p.b.)</td>
<td>Y</td>
<td>$O(n^2)$</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Mattern [14]</td>
<td>non-FIFO</td>
<td>N (p.b.)</td>
<td>Y</td>
<td>$O(n)$</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Acharya-Badrinath [1]</td>
<td>CO</td>
<td>Y</td>
<td>Y</td>
<td>$2n$</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Alagar-Venkatesan [2]</td>
<td>CO</td>
<td>Y</td>
<td>Y</td>
<td>$3n$</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Proposed snapshot</td>
<td>FIFO</td>
<td>Y</td>
<td>Y</td>
<td>$6n$</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Proposed snapshot (w/o channel states)</td>
<td>non-FIFO</td>
<td>Y</td>
<td>Y</td>
<td>$6n$</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Presented: the first version records consistent process states without requiring FIFO channels and without using any form of message send/receive or event counters, whereas the second version records process states and channel states consistently but requires FIFO channels. Critchlow and Taylor have shown that for a system with non-FIFO channels, a snapshot algorithm must either use piggybacking or use inhibition [4]. Hence, the second version of the algorithm cannot be improved upon to also record channel states while retaining the properties of no inhibition and no piggybacking while using non-FIFO channels.

Table 1 compares the proposed algorithms with the existing algorithms. Besides serving as a general-purpose snapshot algorithm, the proposed algorithm can detect any arbitrary stable predicate [3], which was an unsolved problem under the assumptions P1-P4. While many specialized algorithms are tailored to specific stable properties, such as deadlock, termination, and garbage, the following algorithms detect general stable predicates.

- Kshemkalyani-Singhal's two-phase algorithm [9], based on Ho-Ramamooorthy's algorithm [6], showed how to correctly detect deadlocks. A general method to detect stable properties was then outlined [9]. In essence, if a property does not change between the two serial phases of uncoordinated observations, the property must hold at some instant between the two phases. If it is stable, it must hold henceforth. While locally stable predicates can be detected satisfying assumptions (P1)-(P4) and without assuming FIFO channels, details of detecting arbitrary stable predicates are not given.
Neither Marzullo and Sabel [13] nor Schiper and Sandoz [15] showed any relationship between the classes of strong stable and locally stable properties. These existing algorithms can only detect some subclass of stable predicates, and do not satisfy (P1)-(P4). The proposed algorithm can detect any stable predicate.

**Summary of Main Contributions:**

1. The snapshot algorithms we propose for FIFO channels and for non-FIFO channels are linear in the number of messages, and satisfy (P1)-(P4).
2. The non-FIFO version of our snapshot algorithm can be used to detect locally stable predicates, under assumptions (P1)-(P4).
3. The FIFO version of our snapshot algorithm can be used to detect any stable predicate, under assumptions (P1) to (P4).

**System Model:**

An asynchronous execution in a distributed system is modeled as \((E, \prec)\), where \(\prec\) is the causality relation [11] on the set of events \(E\) in the execution. \(E = \bigcup_{i \in N} E_i\), where \(E_i\) is the totally ordered chain of event at process \(P_i\). An event executed by \(P_i\) is denoted \(e_i\). A cut \(C\) is a subset of \(E\) such that the events of a cut are downward-closed within each \(E_i\). A consistent cut is a downward-closed subset of \(E\). The system state after the events in a cut is a global state; if the cut is consistent, the corresponding system state is termed a consistent global state. An execution slice is defined as the difference of two cuts \(D \setminus C\), where \(C \subseteq D\). The slice is also referred to as a suffix of \(C\) with respect to \(D\). When \(D\) is not explicitly specified, its default value is the execution \(E\).

The execution history at a process \(P_i\) is a sequence of alternating states and local events, \((s^{(i)}_0, e^{(i)}_1, s^{(i)}_1, e^{(i)}_2, s^{(i)}_2, e^{(i)}_3, \ldots)\). Events and messages of the snapshot algorithm form a superimposed control execution. Among the application events and messages, those that are relevant to the predicate of interest are the relevant events and messages, respectively. We assume that all events, variables, and messages recorded or logged are the relevant ones.

## 2 The Three-phase Uncoordinated Snapshot Algorithm

The proposed algorithm is inspired by the two-phase deadlock detection algorithm [9]. The main idea of our algorithm is as follows. The algorithm takes three serial uncoordinated ‘snapshots’ that may be inconsistent. A consistent global state that lies between the first and the second inconsistent ‘snapshots’ is computed with the help of the third ‘snapshot’ and some local processing.

Any process can initiate the algorithm which consists of three phases that are serially executed. The algorithm involves some local processing by the initiator. Each phase involves the initiator sending a request to each other process, and then the processes replying to the initiator. The initiator can communicate directly to/from the various processes, or a wave algorithm [5] can be used in conjunction with a superimposed topology such as a ring or a tree. The snapshot algorithm is independent of this detail.
Phase I: The initiator requests the processes to begin recording an execution slice. The global state from which processes begin recording is denoted $Z$, and is represented by the array $Z[1 ... n]$ at the initiator. The local states recorded in $Z$ are not coordinated and may be inconsistent.

Phase II: The initiator then collects the slice of each process execution since the time each process began recording its slice and reported its local state in Phase I, until the time each process chooses to reply to the Phase II request. The local slice of each process that is reported to the initiator is stored by the initiator in array $Slice_A[1 ... n]$. Each process begins to record the next slice, denoted $Slice_B$, after replying to the Phase II request.

Phase III: The initiator then collects the slice of each process execution since the time each process reported its local state in Phase II, until the time each process chooses to reply to the Phase III request. The local slice of each process that is reported to the initiator is stored by the initiator in array $Slice_B[1 ... n]$. Based on $Slice_A$ and $Slice_B$, the initiator computes a consistent global state.

$Slice_B$ is used in two different ways, depending on whether channel states are to be recorded.

- If channel states are not to be recorded and the channels are non-FIFO, $Slice_B$ is used to identify a consistent state within $Slice_A$ by helping to eliminate states in $Slice_A$ that are inconsistent.
- If channel states are also to be recorded and FIFO channels are assumed, then $Slice_B$ is also useful to capture the channel states. In this case, the recording within $Slice_B$ completes at each process when the messages sent by other processes to that process in (and before) $Slice_A$ have been received. This condition is detectable using the control information.
Fig. 2. Six types of events in Slice_A.

distributively sent to the initiator in the Phase II reply messages and then conveyed on the Phase III request received from the initiator.

The possibly inconsistent states collected by the initiator in Z, Slice_A, and Slice_B are illustrated in Figure 1. The initiator computes a consistent global state S, such that Z ⊆ S ⊆ A using Slice_A and Slice_B. Specifically, observe that Z may be inconsistent because messages sent in Slice_A may have been received in Z. Also observe that due to the existence of global time instant \( t_A \) which is any time instant between the last recording of Phase I and the first recording of Phase II, no message sent in Slice_B could have been received before \( t_A \). There exists at least one consistent cut in Slice_A, namely the cut at physical time \( t_A \). However, as the application execution including its messages cannot be modified, and as timestamps are also not used in the algorithm, computing a consistent state \( S \) such that \( Z \subseteq S \subseteq A \) is tricky. In Slice_A, there are six types of events (see Figure 2):

1. send event, for a message that gets delivered in Z
2. send event, for a message that gets delivered in Slice_A
3. send event, for a message that gets delivered after Slice_A
4. receive event, for a message that was sent in Z
5. receive event, for a message that was sent in Slice_A
6. receive event, for a message that was sent after Slice_A

To make Z consistent, we need to add that prefix from Slice_A that contains (i) all events of type (1) and no events of type (6), and (ii) the local states of processes are mutually consistent. Alternately, as \( A \) is also not consistent, \( A \) can be made consistent by subtracting that suffix that contains (i) all events of type (6), and (ii) further events to ensure that the resulting local states of processes are mutually consistent. With either approach, a consistent execution prefix exists, namely, the global state at \( t_A \) which satisfies both sets of conditions. Observe that all events of type (1) precede all events of type (6). Let \( S(t) \) be the prefix of the execution at global time \( t \). We now have the following.
From Figure 1, we have \( S(t_{z_{end}}) \subseteq S(t_A) \subseteq S(t_{A_{start}}) \), where \( t_{A_{start}} \) denotes the time instant of the first local recording of phase II among all the processes, and \( t_{z_{end}} \) denotes the time instant of the last local recording of phase I among all the processes.

Let \( S_{max} \) be \( Z + \) the largest prefix from \( Slice_A \) that does not include an event of type (6). Note that \( S_{max} \) may not be consistent.

A tight lower bound on \( S_{max} \) is the value of \( S(t_{A_{start}}) \). A tight upper bound on \( S_{max} \) is the value of \( S(t_{A_{end}}) \), where \( t_{A_{end}} \) denotes the time instant of the last local recording of phase II among all the processes. Thus, \( S(t_{A_{start}}) \subseteq S_{max} \subseteq A \subseteq S(t_{A_{end}}) \).

The algorithm computes the largest consistent snapshot \( S \) such that \( S(t_{A_{start}}) \subseteq S \subseteq S_{max} \subseteq A \subseteq S(t_{A_{end}}) \), by removing the minimum slice suffix from \( S_{max} \) to get a consistent global state.

The third phase of recordings plays two roles.

- For both versions of the algorithm, \( Slice_B \) helps to identify \( S_{max} \) by identifying messages sent in \( Slice_B \) that were received in \( Slice_A \).
- \( Slice_B \) also helps to identify the in-transit messages by ensuring that all the messages that have been sent up to and including in \( Slice_A \) are received before the recording of the end of \( Slice_B \). This mechanism works only if FIFO channels are assumed.

The two versions of the algorithm are presented together in Figs. 3 and 4.

### 2.1 Consistent State Recording under non-FIFO channels

This section describes a global snapshot algorithm that works with non-FIFO channels and satisfies properties (P1) to (P4). This algorithm records a consistent global state but does not capture channel states.

Figure 3 gives the code for the three-phase processing. The underlined pseudocode and data structures are ignored by this (version of the) algorithm. Step (1) describes the processing followed by the initiator. Step (2) describes the processing at all the nodes. In the first phase, an acknowledgement is sufficient from the nodes to the initiator; the local states are not required to be reported. When \( Slice_A \) and \( Slice_B \) are recorded, observe the following.

- Only the relevant local and send/receive events, of interest to the application or predicate being monitored, are recorded in the log of the slice.
- Messages are not modified with sequence numbers to conform to requirements (P1)-(P4). In addition, no counters for sequence numbers for the messages sent or received, or for the event count, are required at processes. A hash or checksum (\( O(1) \) space) computed on each message sent or received is stored in the log of the slice, to enable matching a message in the sender’s log with the same message in the receiver’s log. No messages are stored.
- Within the log of a slice at a process, sequence numbers are assigned to events. However, these sequence numbers are of significance only to that
process and within the slice. The sequence numbers have no global significance to the processes, or even within a process outside its slice. These numbers are used by the initiator to perform a simple ordering among the process events included in the slice.

After Slice\_A and Slice\_B have been collected at the end of the three phases, the initiator invokes procedure \texttt{Compute\_Consistent\_Snapshot} in Figure 4 to compute a consistent cut $S$ from Slice\_A and Slice\_B. This cut $S$ satisfies $S(t_{A_{\text{start}}}) \subseteq S \subseteq S_{\text{max}} \subseteq A \subseteq S(t_{A_{\text{end}}})$ and is computed by iteratively removing the minimum slice suffix from $S_{\text{max}}$ to get a consistent global state. For convenience, this procedure represents each of the two slices as an array $[1 \ldots n]$ of an array of integers. For example, Slice\_A\_events$[1 \ldots i \ldots n][1 \ldots a_i]$, where Slice\_A\_events$[i][j]$ denotes the $j$th event at process $P_i$, represents Slice\_A.

In Figure 4, lines (3a)-(3d) identify $S_{\text{max}}$. Line (3e) initializes the integer vector variables $S$, $T$, and $U$ to $S_{\text{max}}$. These vectors denote global states but the integer values denoting sequence numbers of the states in the slice have significance local to the initiator only. For example, $S[i]$ was assigned by $P_i$ relative to the start of the slice and represents the sequence number of a local state of $P_i$ in the slice. Lines (3f-3g) initialize the vector variable $V$ to the state at the end of Slice\_A, namely, the state $A$. $S$ is always set to the current known upper bound of the consistent global state that is sought. Vector $V$ denotes the state that is the best known upper bound on the global state such that messages sent in the slice $V \setminus S$ may cause $S$ to be inconsistent. Vectors $T$ and $U$ are working variables used to update $S$ and $V$.

The main loop (3h)-(3q) updates $S$ and $V$ iteratively. In lines (3i)-(3l), $U$ is used to track the prefix of the current $S$ such that there are no inconsistencies caused by messages sent in $V \setminus S$. A message at the sender is matched with the same message at the receiver by comparing their hashes or checksums (line 3k). If the message sent at $\text{Slice\_A\_events}[i][y]$ is received at $\text{Slice\_A\_events}[j][z]$, then $U[j]$ is updated to the minimum of its current value and $z - 1$ (line 3l). An inconsistency, if any, is thus eliminated by removing the minimum suffix from the execution slice for the receiver $P_j$. However, messages sent in the slice $S \setminus U$ may still cause $U$ to be inconsistent; this needs to be tested in the next iteration. Lines (3m)-(3p) initialize the values of $S$, $T$, $U$, and $V$ for the next iteration. The procedure finishes when the loop (3i)-(3l) does not find any inconsistencies in the current value of $S$ in line (3m).

### 2.2 Consistent State and Channel Recording under FIFO Channels

This section presents an enhanced algorithm that also records channel states if channels are FIFO. Figure 3 gives the code for the three-phase processing. Underlined pseudo-code and data structures are also executed by this version of the algorithm. Procedure \texttt{Compute\_Consistent\_Snapshot} in Figure 4 computes a consistent cut from Slice\_A and Slice\_B, and is common to both versions of the algorithm.
(variables at an initiator)
array of states: \( Z[1 \ldots n] \);  // Phase 1 recordings
array of sequence of events: \( \text{Slice}_A[1 \ldots n] \);  // Phase 2 recordings
array of sequence of events: \( \text{Slice}_B[1 \ldots n] \);  // Phase 3 recordings
array of int: \( \text{Global}_\text{Sent}[1 \ldots n, 1 \ldots n] \);  // \( \text{Global}_\text{Sent}[i, j] \equiv \# \text{msgs} P_i \to P_j \)
array of int: \( \text{Global}_\text{Received}[1 \ldots n, 1 \ldots n] \);  // \( \text{Global}_\text{Received}[i, j] \equiv \# \text{messages received by } P_i \text{ from } P_j \)
(array of local events)
array of int: \( \text{Slice}_\text{Log} \);  // log of local events
array of int: \( \text{Sent}[1 \ldots n] \);  // \( \text{Sent}[k] \equiv \# \text{messages sent to } P_k \)
array of int: \( \text{Received}[1 \ldots n] \);  // \( \text{Received}[k] \equiv \# \text{messages received from } P_k \)
array of int: \( \text{Must}_\text{Receive}[1 \ldots n] \);  // \( \text{Must}_\text{Receive}[k] \equiv \# \text{messages to be recd. from } P_k \) before Phase III report

1) Process \( P_{\text{init}} \): initiates the algorithm, where \( 1 \leq \text{init} \leq n \).
   (1a) send Request(Phase 1 Report) to all \( P_j \);
   (1b) await Report(Phase 1 Report) from all processes;
   (1c) (\( \forall j \)) \( Z[j] \leftarrow \text{Phase 1 Report State received from } P_j \);
   (1d) (\( \forall j \)) \( \text{Global}_\text{Received}[j][1 \ldots n] \leftarrow \text{Phase 1 Report Received}[1 \ldots n] \);
   (1e) send Request(Phase 2 Report) to all \( P_j \);
   (1f) await Report(Phase 2 Report) from all processes;
   (1g) (\( \forall j \)) \( \text{Slice}_A[j] \leftarrow \text{Phase 2 Report} \text{Slice}_\text{Log} \) received from \( P_j \);
   (1h) (\( \forall j \)) \( \text{Global}_\text{Sent}[j][1 \ldots n] \leftarrow \text{Phase 2 Report} \text{Sent}[1 \ldots n] \) from \( P_j \);
   (1j) send to all \( P_j \)
   \( \text{Request(Phase 3 Report)} + \text{Global}_\text{Sent}[1 \ldots n][j] \) piggybacked;
   (1j) await Report(Phase 3 Report) from all processes;
   (1k) (\( \forall j \)) \( \text{Slice}_B[j] \leftarrow \text{Phase 3 report received from } P_j \);
   (1l) \( S \leftarrow \text{Compute Consistent Snapshot} (\text{Slice}_A, \text{Slice}_B) \);
   (1m) \text{Compute In-transit Messages}(S).

2) Process \( P_j \): executes the following, where \( 1 \leq j \leq n \).
   (2a) On receiving Request(Phase 1 Report) from \( P_{\text{init}} \),
   (2b) send local state and \( \text{Received}[1 \ldots n] \) in Report(Phase 1 Report) to \( P_{\text{init}} \);
   (2c) Begin recording log of events in \( \text{Slice}_\text{Log} \);
   (2d) On receiving Request(Phase 2 Report) from \( P_{\text{init}} \),
   (2e) send \( \text{Slice}_\text{Log} \text{ and } \text{Sent}[1 \ldots n] \) in Report(Phase 2 Report) to \( P_{\text{init}} \);
   (2f) Reset \( \text{Slice}_\text{Log} \);
   (2g) On receiving Request(Phase 3 Report) and \( \text{Must}_\text{Receive}[1 \ldots n] \) from \( P_{\text{init}} \),
   (2h) Await until, (\( \forall k \)) \( \text{Received}[k] \geq \text{Must}_\text{Receive}[k] \);
   (2i) send \( \text{Slice}_\text{Log} \text{ in Report(Phase 3 Report) to } P_{\text{init}} \);
   (2j) Stop recording events in \( \text{Slice}_\text{Log} \).

**Fig. 3.** Three-phase algorithm to record a global snapshot. Underlined code is executed if channel states are needed in a FIFO system.

Procedure \text{Compute In-transit Messages} is used to compute the channel states. This procedure requires the data structures \( Z[1 \ldots n] \) and integer arrays \( \text{Global}_\text{Sent}[1 \ldots n, 1 \ldots n] \) and \( \text{Global}_\text{Received}[1 \ldots n, 1 \ldots n] \) at the initiator during the processing of the algorithm. Integer vectors \( \text{Received}[1 \ldots n] \),
(3) Process $P$ executes $\text{Compute} \_\text{Consistent} \_\text{Snapshot}(\text{Slice}_A, \text{Slice}_B)$.

array of int: $S_{max}, S, T, U, V$;
array of array of int: $\text{Slice}_A\_\text{events}[1 \ldots i \ldots n][1 \ldots a]$;
array of array of int: $\text{Slice}_B\_\text{events}[1 \ldots i \ldots n][1 \ldots b]$;

// Alternate representation of slices

(3a) for $i = 1$ to $n$ do
    (3b) if $\text{Slice}_A[i][x + 1]$ is the $1^{st}$ receive in $\text{Slice}_A[i]$ of a msg. sent in $\text{Slice}_B$ then
    (3c) $S_{max}[i] \leftarrow x$
    (3d) else $S_{max}[i] \leftarrow a$;
    (3e) $S, T, U \leftarrow S_{max}$;
    (3f) for $i = 1$ to $n$ do
        (3g) $V[i] \leftarrow a$;
    (3h) repeat
    (3i) for $i = 1$ to $n$ do
        (3j) for $y = T[i] + 1$ to $V[i]$ do
        (3k) if message($\text{Slice}_A\_\text{events}[i][y], \text{Slice}_A\_\text{events}[j][z]$) then
            (3l) $U[z] \leftarrow \min(U[j], z - 1)$; // modify $U$ to make it consistent
        (3m) if $T = U(= S)$ then return($S$);
        (3n) $S \leftarrow \min(T, U)$; // $S$ is current upper bound on consistent state
        (3o) $V \leftarrow \max(T, U)$; //current upper bound on source of inconsistency
        (3p) $T, U \leftarrow S$;
    (3q) forever.

(4) Process $P$ executes $\text{Compute} \_\text{In-transit} \_\text{Messages}(S)$.

(4a) $(\forall j) \text{transit}(S[i], S[j]) \leftarrow \emptyset$;
(4b) $(\forall i)$ compute $\text{Global} \_\text{Sent}[i, 1 \ldots n]$ for $S[i]$ using $\text{Slice}_A, \text{Global} \_\text{Sent}[i, 1 \ldots n]$;
(4c) for $j = 1$ to $n$ do
    (4d) for each successive event $e_j$ in $\text{Slice}_A[j][1 \ldots a_j]$ and $\text{Slice}_B[j][1 \ldots b_j]$ do
        (4e) if a message $M$ was received from $i$ (at this event with seq. $\neq x$) then
            (4f) $\text{Global} \_\text{Received}[j, i] \leftarrow +$
        (4g) if $\text{Global} \_\text{Sent}[i, j] \geq \text{Global} \_\text{Received}[j, i]$ and $x > S[j]$ then
            (4h) $\text{transit}(S[i], S[j]) \leftarrow \text{transit}(S[i], S[j]) \cup \{M\}$.

Fig. 4. Finding a consistent state iteratively, and computing in-transit messages. Underlined code is executed if channel states are needed in a FIFO system.

$\text{Sent}[1 \ldots n]$, and $\text{Must} \_\text{Receive}[1 \ldots n]$ must also be maintained at each node. $\text{Sent}[j]$ and $\text{Received}[j]$ track the count of the number of messages sent to and received from process $P_j$, respectively. The main idea is simple. The state of channel $(P_i, P_j)$ in a global state containing local states $S[i]$ and $S[j]$ at the processes, denoted as $\text{transit}(S[i], S[j])$, is simply those messages sent by $P_i$ till state $S[i]$ that are not received until state $S[j]$ at $P_j$. One important difference from the previous version is that sequence numbers used to count events at a process are not local to a slice, but local to the entire execution of that process. This is to capture in-transit messages for channel states. Such messages could have been sent before $\text{Slice}_A$ and need to be detected. To compute the channel
Table 2. Complexity of the proposed non-inhibitory nonintrusive snapshot algorithm. Both \texttt{Slice}_A and \texttt{Slice}_B are thin slices, and their expected size is the same.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Recording snapshot (non-FIFO, no channel states)</th>
<th>Recording snapshot + channel states (w/ FIFO channels)</th>
</tr>
</thead>
<tbody>
<tr>
<td># messages</td>
<td>6(n−1)</td>
<td>6(n−1)</td>
</tr>
<tr>
<td>Msg. space (total)</td>
<td>(O(</td>
<td>\texttt{Slice}_A</td>
</tr>
<tr>
<td>Time complexity (initiator)</td>
<td>(O(</td>
<td>\texttt{Slice}_A</td>
</tr>
<tr>
<td>Time complexity (non-initiator)</td>
<td>(O(</td>
<td>\texttt{Slice}_A</td>
</tr>
<tr>
<td>Space complexity (initiator)</td>
<td>(O(</td>
<td>\texttt{Slice}_A</td>
</tr>
<tr>
<td>Space complexity (non-initiator)</td>
<td>(O(1/n \cdot</td>
<td>\texttt{Slice}_A</td>
</tr>
<tr>
<td>Properties</td>
<td>No inhibition</td>
<td>No inhibition</td>
</tr>
<tr>
<td></td>
<td>App. messages unmodified</td>
<td>App. messages unmodified</td>
</tr>
<tr>
<td></td>
<td>execution unmodified</td>
<td>execution unmodified</td>
</tr>
<tr>
<td></td>
<td>no log of history</td>
<td>no log of history</td>
</tr>
<tr>
<td></td>
<td>+ \texttt{Sent}, Receive_\texttt{vecs}/process +</td>
<td>\texttt{Sent}, \texttt{Receive} at init</td>
</tr>
</tbody>
</table>

state while satisfying conditions (P1) – (P4) and specifically that no sequence numbers can be tagged on messages, three issues need to be addressed.

1. Messages sent by \(P_i\) to \(P_j\) before state \(S[i]\) must have reached \(P_j\) by the end of \texttt{Slice}_B.

   This is ensured by using the local \texttt{Sent} vector at each process and the \texttt{Global\_Sent} array at the initiator. In the Phase II recording reported to the initiator, the \texttt{Sent} vectors reported (line (2e)) are used to populate \texttt{Global\_Sent} (line (1h)). The Phase III request sent to each process contains the piggybacked information about how many messages have been sent to that process by other processes (line (1i)). A process postpones its Phase III recording of the end of \texttt{Slice}_B until all these number of messages, remembered in \texttt{array Must\_Receive}, have been delivered locally (line (2h)).

2. The set of messages sent by \(P_i\) to \(P_j\) up to the snapshot state \(S[i]\), denoted here as \(X\), should be identifiable. There are two parts to this.

   - This set contains all the messages received by \(P_j\) from \(P_i\) with sequence numbers less than the value of \texttt{Sent}[j] at \(S[i]\). This value of \texttt{Sent}[j] at \(S[i]\) is computed (line (4b)) using \texttt{Global\_Sent}, constructed from the Phase II report, and working backwards using the log \texttt{Slice}_A, also reported in Phase II. The resulting message count (i.e., the value of \texttt{Sent}[j] at \(S[i]\)) is stored in-situ in the data structure \texttt{Global\_Sent}[i,j] as it is updated.

   - The messages received by \(P_j\) are enumerated as per the sequence numbers assigned by \(P_i\), in lines (4d)-(4e). The enumeration of the sequence numbers is done in lines (4c)-(4f) using \texttt{Global\_Received}, reported in
Table 3. Comparing algorithms to detect stable predicates.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Detectable</td>
<td>locally stable</td>
<td>strong stable</td>
<td>locally stable</td>
<td>all stable</td>
</tr>
<tr>
<td>predicates</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>overhead at</td>
<td>vector clock, $O(n)$</td>
<td>vector clock, $O(n)$</td>
<td>–</td>
<td>$Sent/Received$</td>
</tr>
<tr>
<td>nodes (=</td>
<td>+ entire log</td>
<td>+ entire log</td>
<td>event log in</td>
<td>$(O(n)) + event$</td>
</tr>
<tr>
<td>control msg</td>
<td>of msgs &amp; events</td>
<td>of msgs &amp; events</td>
<td>slice during</td>
<td>&amp; log in slice</td>
</tr>
<tr>
<td>overhead)</td>
<td>w/timestamps</td>
<td>w/timestamps</td>
<td>3-phase</td>
<td>during 3-phase</td>
</tr>
<tr>
<td>App. msg</td>
<td>vector clock, $O(n)$</td>
<td>vector clock, $O(n)$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>processing</td>
<td>by initiator</td>
<td>by initiator</td>
<td>by initiator</td>
<td>by initiator</td>
</tr>
<tr>
<td>Channels</td>
<td>FIFO</td>
<td>non-FIFO</td>
<td>non-FIFO</td>
<td>FIFO</td>
</tr>
<tr>
<td># control</td>
<td>$(n - 1)$</td>
<td>$(n - 1)$</td>
<td>$6(n - 1)$</td>
<td>$6(n - 1)$</td>
</tr>
<tr>
<td>messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Phase I to the initiator (line (1d)), and working forwards using the log $\text{Slice}_A$ reported in Phase II and the log $\text{Slice}_B$ reported in Phase III.

The enumeration is done in-situ in $\text{Global}_{\text{Received}}[j, i]$.

If $\text{Global}_{\text{Received}}[j, i] \leq \text{Global}_{\text{Sent}}[i, j]$ for a message, then that message belongs to $\mathcal{X}$.

3. The set of messages received by $P_j$ from $P_i$ after the snapshot state $S[j]$, denoted here as $\mathcal{Y}$, should be identifiable.

These are the messages received from $P_i$ at $P_j$ in states numbered $x$ such that $x > S[j]$.

From (2) and (3) above, $transit(S[i], S[j]) = \mathcal{X} \cap \mathcal{Y}$, is expressible as $\{M \text{ rec'd by } P_j \text{ at event } x \mid \text{Global}_{\text{Received}}[j, i] \leq \text{Global}_{\text{Sent}}[i, j] \land x > S[j]\}$.

Unlike the algorithm in Section 2.1, $\text{Slice}_{\text{Log}}$, $\text{Slice}_A$, $\text{Slice}_B$ record messages for send and receive events if the contents of in-transit messages are required. The pseudo-code data structures do not reflect this for simplicity.

3 Complexity

The complexity analysis assumes a flat tree topology with the initiator as the root. A similar analysis can be conducted for the ring and more general tree topologies. Table 2 summarizes the complexity results. Note that the slices $\text{Slice}_A$ and $\text{Slice}_B$ are both thin slices, and their expected size is the same. Hence, the complexity is expressed in terms of $\text{Slice}_A$ only. The expected width of $\text{Slice}_A$ is the execution log that occurs in $\text{rtt}_{\text{max}}$, the expected round-trip time between the two furthest nodes in the network. Let $\text{max}(\text{out}_A)$ denote the maximum time for a message sent in $\text{Slice}_A$ to reach its destination. The expected width of $\text{Slice}_B$ is $\text{max}(\text{rtt}_{\text{max}}, \text{out}_A)$ which is also $\text{rtt}_{\text{max}}$. 
4 Detecting Stable Predicates

The proposed algorithm to record a consistent global state can be used to detect any stable predicate. (See [7] for details.) Each process records the (possibly changing) values of the variables over which the predicate is defined, in Store_A. When the initiator computes the consistent state S, it can also evaluate the predicate over these variables in state S. If the predicate is evaluated as true, then it is true and remains true henceforth because it is stable.

– Version 1 can detect locally stable predicates predicates.
– Version 2 can detect any stable predicate.

Table 3 compares the features and the complexities of the proposed algorithm with those of the algorithms by Marzullo-Sabel [13] and Schiper-Sandoz [15].

The full version of the results of this paper, including the correctness proofs and the complexity analysis, is in [7].

References